**JavaScript**

1. **What are closures in JavaScript? Can you provide an example?**

Closures are functions that retain access to variables from their containing (enclosing) scope even after the parent function has finished executing.

**Example:**

function outerFunction() {

let outerVariable = 'I am outer';

return function innerFunction() {

console.log(outerVariable);

}

}

let innerFunc = outerFunction();

innerFunc(); // Output: I am outer

1. **Explain the concept of prototypal inheritance in JavaScript.**

In JavaScript, every object has a prototype property, which allows object properties and methods to be inherited. Prototypal inheritance enables objects to inherit properties and methods from other objects.

**Example:**

function Person(name) {

this.name = name;

}

Person.prototype.greet = function() {

console.log('Hello, my name is ' + this.name);

};

let john = new Person('John');

john.greet(); // Output: Hello, my name is John

1. **What are the differences between null and undefined in JavaScript?**

**null:** Think of null as an intentional absence of any value. It's like an empty box that a programmer sets to indicate that there is no value present. So, when you see null, it means that there is explicitly no value assigned to something.

**undefined:** On the other hand, undefined means that a variable has been declared but hasn't been assigned any value yet. It's like having a box that hasn't been filled with anything yet, not even with null. It's the default value for uninitialized variables.

**Example:**

let x = null;

let y;

1. **Explain event delegation in JavaScript.**

Event delegation is a technique where you add a single event listener to a parent element rather than multiple event listeners to individual child elements. This is achieved by utilizing event bubbling. Events that occur in the child elements are then handled by the parent, based on the event target. This improves performance and reduces memory consumption, especially in cases where there are many child elements.

Let's imagine a scenario where you have a list of 100 items and you want to perform a specific action when any item is clicked. A common approach might be to attach an event listener to each item. However, this would be resource-intensive and inefficient, especially if the list grows dynamically. Instead, we can attach a single event listener to the parent element and use event delegation to handle clicks on individual items. This not only improves performance but also ensures that dynamically added elements are covered.

In event delegation, we leverage the bubbling phase. During this phase, the event starts from the target element and travels up the DOM tree. This means that a click event on a child element will also be registered on its parent element, grandparent element, and so on, till it reaches the root document object.

1. **What is event propagation in JavaScript?**

Event propagation in JavaScript refers to the mechanism by which events propagate or travel through the DOM (Document Object Model) hierarchy. There are two main phases of event propagation: capturing phase and bubbling phase.

**Capturing Phase:** In this phase, the event is captured by the outermost element and propagates down through its descendants to the target element. This phase happens before the actual target element receives the event.

**Target Phase:** Once the event reaches the target element, it triggers the event listeners attached directly to that element.

**Bubbling Phase:** After the target element has handled the event, the event starts bubbling up from the target element through its ancestors back to the outermost element. Event listeners attached to ancestor elements can intercept the event during this phase.

1. **What are the different ways to create objects in JavaScript?**

* **Object literal syntax:**

let obj = {

key: value

};

* **Constructor functions:**

function Person(name) {

this.name = name;

}

* **Using the Object.create() method:**

let obj = Object.create(proto);

* **ES6 classes:**

class Person {

constructor(name) {

this.name = name;

}

}

1. **What is the event loop in JavaScript, and how does it work?**

The event loop is the process that handles asynchronous callbacks in JavaScript. It continuously checks the call stack and the task queue. When the call stack is empty, it takes the first callback from the task queue and pushes it onto the call stack for execution. This process continues indefinitely, allowing asynchronous operations to be handled in a non-blocking manner.

The Event Loop is a mechanism in JavaScript that handles asynchronous operations. It continuously checks the call stack and the task queue. When the call stack is empty, it takes the first callback from the task queue and pushes it onto the call stack for execution. This process allows JavaScript to be non-blocking and handle asynchronous operations efficiently.

1. **What is the difference between == and === in JavaScript?**

== is the equality operator that performs type coercion, meaning it converts the operands to the same type before making the comparison.

=== is the strict equality operator that does not perform type coercion, so it only returns true if both the value and the type of the operands are the same.

1. **Explain the concepts of hoisting in JavaScript.**

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compilation phase. However, only the declarations are hoisted, not the initializations.

**For example:**

console.log(x); // undefined

var x = 5;

----

var x;

console.log(x); // undefined

x = 5;

1. **What are the differences between arrow functions and regular functions in JavaScript?**

Arrow functions are a concise syntax for writing function expressions. The key differences include:

* Arrow functions do not have their own this context; they inherit this from the surrounding code.
* Arrow functions do not have their own arguments object; arguments refers to the arguments of the enclosing scope.
* Arrow functions cannot be used as constructors and do not have their own prototype.

1. **Explain the concept of Promises in JavaScript and how they differ from callbacks.**

Promises are objects representing the eventual completion or failure of an asynchronous operation. They provide a cleaner alternative to traditional callback-based approaches for handling asynchronous code. Promises have built-in methods like **then()** and **catch()** to handle success and failure cases, making asynchronous code more readable and easier to manage compared to callbacks.

1. **What are some common techniques to improve the performance of JavaScript code?**

Some common techniques include:

* Minifying and compressing JavaScript files.
* Using asynchronous programming patterns like Promises or async/await to avoid blocking the main thread.
* Debouncing and throttling event handlers to reduce the frequency of function executions.
* Implementing efficient data structures and algorithms.
* Optimizing DOM manipulation and reducing reflows and repaints.
* Caching frequently accessed data and avoiding unnecessary network requests.

1. **What is the difference between function declaration and function expression in JavaScript?**

Function declaration and function expression are two ways to define functions in JavaScript, but they have some differences in terms of hoisting and when they are available for use in the code.

1. **Function Declaration:**

* Function declarations are defined using the function keyword followed by the function name and the function body.
* Function declarations are hoisted, meaning they are available for use throughout the scope in which they are defined, even before the actual function declaration in the code.

**Example:**

function greet() {

console.log('Hello!');

}

1. **Function Expression:**

* Function expressions involve defining a function as part of an expression, usually by assigning it to a variable.
* Function expressions are not hoisted, meaning they are only available for use after they have been defined in the code.

**Example:**

const greet = function() {

console.log('Hello!');

};

Here's a brief comparison between the two:

* **Function Declaration:**

**Hoisted:** Yes

**Syntax:** function functionName() { ... }

**Usage:** Can be used before its declaration in the code.

* **Function Expression:**

**Hoisted:** No

**Syntax:** const functionName = function() { ... };

**Usage:** Can only be used after its declaration in the code.

1. **What are the differences between the let, const, and var keywords in JavaScript?**

**var:** Variables declared with var are function-scoped (or globally scoped if declared outside of a function). They can be redeclared and reassigned.

**let:** Variables declared with let are block-scoped, meaning they are only accessible within the block they are defined in. They can be reassigned but not redeclared within the same scope.

**const:** Variables declared with const are also block-scoped and cannot be reassigned once initialized. However, for objects and arrays declared with const, their properties or elements can be modified.

1. **Explain the difference between the map() and forEach() methods in JavaScript arrays.**

* Both **map()** and **forEach()** are methods available on JavaScript arrays for iterating over elements.
* **forEach():** It iterates over each element in the array and executes a callback function for each element. It doesn't return a new array.
* **map():** It iterates over each element in the array, applies a callback function to each element, and returns a new array containing the results of the callback function applied to each element.

1. **What are some methods for handling asynchronous operations in JavaScript, other than Promises and Callbacks?**

Apart from Promises and Callbacks, some methods for handling asynchronous operations include:

* **Async/await:** Introduced in ES2017, async/await is a syntactic sugar built on top of Promises. It allows writing asynchronous code in a more synchronous style, making it easier to read and maintain.
* **Event emitters:** Using event-driven programming, you can create custom events and listeners to handle asynchronous tasks.
* **Generators:** Generators allow pausing and resuming the execution of functions, which can be useful for handling asynchronous tasks sequentially.

1. **What are the differences between call(), apply(), and bind() methods in JavaScript?**

* **call():** Invokes a function with a specified this value and arguments provided individually.
* **apply():** Invokes a function with a specified this value and arguments provided as an array or array-like object.
* **bind():** Creates a new function that, when called, has its this value set to a specific value, and prepends any provided arguments to the original function's arguments.

1. **Explain the concept of event bubbling and event capturing in the DOM.**

Event bubbling and event capturing are two mechanisms for handling events in the DOM. Event bubbling occurs when an event is triggered on an element and then bubbles up through its ancestors, executing event handlers on each ancestor. Event capturing, on the other hand, is the opposite; the event starts at the outermost ancestor and moves down to the target element. Both mechanisms provide ways to handle events on multiple elements.

1. **What is the event.preventDefault() method used for in JavaScript event handling?**

The **event.preventDefault()** method is used to prevent the default behavior associated with an event from occurring. For example, when handling a form submission, calling **event.preventDefault()** prevents the form from being submitted to the server, allowing you to handle the form submission using JavaScript.

1. **Explain the concept of memoization and how it can be implemented in JavaScript.**

Memoization is an optimization technique used to cache the results of expensive function calls and return the cached result when the same inputs occur again. This helps reduce redundant computation and improve performance. In JavaScript, memoization can be implemented using closure and a data structure like an object or a Map to store the cached results.

1. **What are the differences between the ES6 class syntax and the traditional prototype-based inheritance in JavaScript?**

The ES6 class syntax provides a more familiar and concise way to define classes and inheritance in JavaScript compared to the traditional prototype-based approach. However, under the hood, ES6 classes still use prototypes for inheritance. Some differences include:

* ES6 classes provide a cleaner syntax for defining constructors, methods, and inheritance.
* With ES6 classes, the extends keyword is used to establish inheritance between classes.
* ES6 classes support the constructor method for initializing object instances.

1. **Explain the concept of the "this" keyword in JavaScript and how its value is determined in different contexts.**

The this keyword in JavaScript refers to the context in which a function is called. Its value is determined dynamically based on how the function is invoked:

* In the global context or inside a function called without an explicit context, this refers to the global object (e.g., window in a browser).
* In the context of an object method, this refers to the object that owns the method.
* When using the new keyword to create an instance of a constructor function, this refers to the newly created instance.

1. **Explain the concepts of currying and partial application in JavaScript. Provide examples for each.**

* **Currying:** Currying is the process of converting a function with multiple arguments into a series of nested functions, each taking a single argument. This allows you to create specialized functions by partially applying arguments.

function multiply(a) {

return function(b) {

return a \* b;

};

}

const multiplyByTwo = multiply(2);

console.log(multiplyByTwo(3)); // Output: 6

* **Partial Application:** Partial application is similar to currying but involves fixing a predefined number of arguments ahead of time.

function add(x, y, z) {

return x + y + z;

}

const addTwoNumbers = add.bind(null, 2);

console.log(addTwoNumbers(3, 4)); // Output: 9

1. **What is the difference between setTimeout() and setInterval() functions in JavaScript?**

* **setTimeout():** Executes a function or evaluates an expression after a specified delay (in milliseconds) and only runs once.
* **setInterval():** Executes a function or evaluates an expression repeatedly at a specified interval (in milliseconds) until it is cleared with clearInterval().

1. **What are the differences between the map(), filter(), and reduce() array methods in JavaScript?**

* **map():** Iterates over an array and applies a function to each element, returning a new array with the results.
* **filter():** Iterates over an array and returns a new array containing only the elements that satisfy a specified condition.
* **reduce():** Applies a function to each element of an array and reduces it to a single value. It takes an accumulator and the current value as arguments and returns the accumulated result.

1. **Explain the concepts of immutability and mutability in JavaScript, and why immutability is important in functional programming.**

* **Immutability:** Immutability refers to the inability of an object to be modified after it is created. In JavaScript, primitive data types (e.g., strings, numbers) are immutable, while objects and arrays are mutable.
* **Mutability:** Mutability refers to the ability of an object to be modified after it is created. Objects and arrays in JavaScript are mutable, meaning their values can be changed.
* Immutability is important in functional programming because it ensures predictable behavior and helps avoid unintended side effects. Immutable data structures are safer to work with in concurrent or parallel environments and facilitate better code readability and debugging.

1. **What is dependency injection in JavaScript?**

Dependency injection (DI) is a design pattern commonly used in software development, including JavaScript, to facilitate loosely coupled code and improve testability and maintainability. In JavaScript, dependency injection involves injecting dependent objects or services into a class or function from the outside, rather than creating them within the class or function itself.

Here's an example to illustrate dependency injection in JavaScript:

Suppose we have a UserService class that depends on a UserRepository for data access. Instead of creating an instance of UserRepository inside the UserService, we inject it as a dependency.

// UserRepository.js

class UserRepository {

constructor() {

// Initialization logic

}

getUsers() {

// Return users from the database

return ['User 1', 'User 2', 'User 3'];

}

}

// UserService.js

class UserService {

constructor(userRepository) {

this.userRepository = userRepository;

}

getAllUsers() {

// Use userRepository to get users

return this.userRepository.getUsers();

}

}

// index.js (or any other entry point)

const UserRepository = require('./UserRepository');

const UserService = require('./UserService');

// Create instances and inject dependencies

const userRepository = new UserRepository();

const userService = new UserService(userRepository);

// Use userService to get users

const users = userService.getAllUsers();

console.log(users); // Output: ['User 1', 'User 2', 'User 3']

1. **What is difference between promises and observable?**

Promises and observables are both constructs used for handling asynchronous operations in JavaScript. While they serve similar purposes, there are some key differences between the two:

**Single vs. Multiple Values:**

* **Promises:** Promises represent a single future value or a single asynchronous operation. Once a promise is resolved with a value, it cannot be changed or updated.
* **Observables:** Observables represent a stream of multiple values over time. They can emit zero or more values asynchronously, and they can be cancelled or unsubscribed from.

**Eager vs. Lazy Execution:**

* **Promises:** Promises are eager, meaning they start executing as soon as they are created, regardless of whether there is a consumer (.then() or .catch()) attached to them.
* **Observables:** Observables are lazy, meaning they only start emitting values when they are subscribed to. This allows for more efficient resource usage, especially when dealing with streams of data.

**Cancellation:**

* **Promises:** Promises do not support cancellation out of the box. Once a promise is initiated, it will resolve or reject with a value, and it cannot be cancelled.
* **Observables:** Observables support cancellation through the **unsubscribe()** method. When an observable is unsubscribed from, it stops emitting values and releases any resources associated with it.

**Error Handling:**

* **Promises:** Promises handle errors using the **.catch()** method. Errors thrown within a promise chain can be caught and handled using **.catch().**
* **Observables:** Observables handle errors using the error callback function passed to the **subscribe()** method. They also have operators like **catchError()** for handling errors within the stream.

**Composition:**

* **Promises:** Promise chaining is the primary way of composing asynchronous operations with promises. Each **.then()** or **.catch()** returns a new promise, allowing for sequential execution of asynchronous operations.
* **Observables:** Observables provide powerful composition operators like **map**, **filter**, **reduce**, and others, which can be used to transform, filter, combine, and manipulate the emitted values in a declarative way.

In summary, while promises are suitable for handling single asynchronous operations with a single value, observables are more powerful and flexible for handling streams of data with multiple values over time, supporting lazy evaluation, cancellation, and advanced composition operators.

1. **What are Cookies, Local Storage and Session Storage? Explain with example.**

Cookies, local storage, and session storage are three mechanisms in web browsers used for storing data locally on the client side. Each has its own characteristics and use cases.

**Cookies:**

* Cookies are small pieces of data stored in the browser that are sent back to the server with subsequent requests.
* Cookies have a maximum size limit (typically 4KB) and can store data as key-value pairs.
* Cookies can be set with an expiration date, after which they are automatically deleted.
* Cookies are primarily used for session management, user authentication, and tracking user behavior.

**Example:**

// Setting a cookie

document.cookie = "username=John Doe; expires=Thu, 18 Feb 2025 12:00:00 UTC; path=/";

// Reading a cookie

const cookies = document.cookie.split('; ');

cookies.forEach(cookie => {

const [name, value] = cookie.split('=');

console.log(name, value);

});

**Local Storage:**

* Local storage is a feature of modern web browsers that allows web applications to store data locally without expiration dates.
* Local storage provides a larger storage capacity (typically 5MB per domain) compared to cookies.
* Data stored in local storage persists even after the browser is closed and reopened.
* Local storage data is accessible only to the same domain that stored it.

**Example:**

// Storing data in local storage

localStorage.setItem('username', 'John Doe');

// Retrieving data from local storage

const username = localStorage.getItem('username');

console.log(username);

**Session Storage:**

* Session storage is similar to local storage but scoped to a particular browsing session.
* Data stored in session storage is cleared when the browsing session ends (i.e., when the browser window is closed).
* Session storage provides a larger storage capacity compared to cookies but is limited to the current browsing session.
* Like local storage, session storage data is accessible only to the same domain that stored it.

**Example:**

// Storing data in session storage

sessionStorage.setItem('token', 'abc123');

// Retrieving data from session storage

const token = sessionStorage.getItem('token');

console.log(token);

In summary, cookies, local storage, and session storage are all used for storing data locally on the client side, but they differ in terms of capacity, persistence, and scope.